7-2 Project Two

Jonathan C. Sanchez

June 29, 2024

Testing is a crucial part of the Software Development Lifecycle (SDLC). When people think about testing software, they often only think it is used to detect bugs in software. However, software testing is much more than detecting bugs, it is used to verify and validate the customer’s requirements, and ensure software is efficient and secure. The test cases written in Project One are used to verify and validate the quality of the implemented requirements provided for the Contact Service, Task Service, and Appointment Service.

The approach taken in Project One was to understand the requirements and determine the logic needed for the implementation of said requirements. As well as ensure that proper coding techniques were followed during implementation through testing. When writing the test cases I often referred to the requirements to write the separate JUnit tests and to ensure that each requirement was properly met. In the testing environments, I used the ‘@Test’ annotation to tell JUnit to run the method(s) as separate test cases. When the tests are initiated, JUnit will construct a new instance of the class and invoke the annotated methods. If the test fails, JUnit will return an exception. For the test to be considered a pass, it will not return an exception. The approach to understand and properly implement the requirements allowed me to implement effective Test Cases that provided relatively high coverage.

The Appointment Service package had an overall Test Coverage of 81.15%. The Contact Service package had an overall Test Coverage of 80.17%, and the Task Service Package had a 79.55% Test Coverage. The overall Test Coverage for all Test Cases and packages is 80.30%.

A screenshot of a service

Description automatically generated

Running the tests with coverage provides information that allows me to refine or change my approach and test cases. If my tests do not have enough coverage I can navigate to specific functions and see what statements are covered and which are not. Knowing which statements do not have coverage allows me to write a new test case to handle/test the uncovered statements or refine previously written cases. Running the JUnit Tests with coverage ensures that the requirements are efficiently implemented without errors or issues.

Technically sound and efficient code mitigates risk and ensures the quality of the software. I ensured that my code was technically sound by implementing error handling similar to that below for all the specified boundaries in the Appointment, Task, and Contact classes.

if (appointmentId == null || appointmentId.length() > 10) {

throw new IllegalArgumentException("Invalid Appointment Id");

}

A second way that I ensure the code is technically sound is by writing test cases to validate the logic and processes used to meet the requirements.

@*Test*

*public* void testAddContact() {

ContactService contactService = new ContactService();

Contact contact = new Contact("12345", "Jonathan", "Sanchez", "5555555555", "123 Main St");

*/\*Check that the contact has been added, the test should return true if it has been added\*/*

assertTrue(contactService.addContact(contact));

assertTrue(contact.getContactID().equals("12345"));

assertTrue(contact.getFirstName().equals("Jonathan"));

assertTrue(contact.getLastName().equals("Sanchez"));

assertTrue(contact.getPhoneNumber().equals("5555555555"));

assertTrue(contact.getAddress().equals("123 Main St"));

*//Verify that the contact exists in the list*

assertTrue(contactService.list.contains(contact));

*/\*Check adding the same contact with the same contactID again. The test should return false if the requirement was properly implemented \*/*

assertFalse(contactService.addContact(contact));

}

Lastly, I mimicked patterns across the various classes to make the code more maintainable. To ensure efficiency, I wrote logical statements that follow optimal processes.

*public* boolean addContact(Contact contact) {

boolean didAdd = false;

if (list.size() == 0 || (!contact.getContactID().equalsIgnoreCase(contact.getContactID())) ) {

list.add(contact);

didAdd = true;

} else {

for (Contact t : list) {

if (contact.getContactID().equalsIgnoreCase(t.getContactID())) {

return didAdd;

}

}

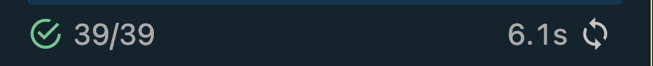
}

System.out.println(didAdd);

return didAdd;

}

I also monitor the size of the project and the time it takes to run a process.

 A blue fabric with white lines

Description automatically generated

Many forms of testing software exist, and one cannot always be certain that a single form of testing will provide enough coverage and assurance. Therefore, for Project One, I conducted four different types of tests. Unit Testing, Functionality Testing, Compilation Testing and Performance Testing. Unit testing allowed me to test individual functions, classes, and methods. Functionality Testing allowed me to test an entire process, such as adding and deleting objects. Compilation Testing tests for compilation errors and ensures variables are not misspelled or misused. The last form of testing I used was Performance Testing. I monitored the size of the project and the time taken to compile it. Examples of testing I did not use are Load Testing, Security Testing, and Compatibility Testing. Load Testing exerts a large demand on the software to monitor its performance. Security Testing identifies vulnerabilities and risks that could affect the user or software. Lastly, Compatibility Testing ensures the software can operate on different Operating Systems, browsers, or networks. Utilizing numerous test types can help reduce risk and assure quality when deploying software.

Testers play a vital role in the Software Development Lifecycle. As I worked through Project One, and the course I had to take on an entirely new mindset. I had to develop a cautious and technical mindset during this course and for Project One to ensure I wrote test cases with good coverage that could identify potential bugs/issues and confirm that the requirements were implemented. During this project I took on the role of a developer and the tester, so I knew exactly what the code written was. However, testers do not always have the opportunity to oversee all parts of development. Therefore, testers must be able to read the code and understand the logic and relationships to write proper test cases. The snippets above show that I used conditional statements in my program. As a tester, I had to understand the conditions to write test cases that could test each condition. A tester with a cautious and technical mindset can change the quality and outcome of a project.

Having a positive or negative bias can affect the quality of software. In elementary school, I had to grade my assignments or my peers. One can imagine what the outcome of that power was. With that being said, I had to approach Project One from an outsider’s perspective. As a developer, I get attached to the code I write and can become protective of it. Therefore, as a tester, I had to keep an open mind and understand that if I found any issues, I would have to think of a new approach and reevaluate the requirements and code.

Companies and stakeholders take on a lot of risk when they employ software development teams. They do not always know the work ethic, morals, and motives of the person they hire. A software engineer can produce defective and vulnerable software that can cause a company and its customers to be at risk of attacks, data leaks/theft, damage to reputation, and financial losses. Software engineers may not always know they are creating vulnerabilities, but some do this knowingly and maliciously. Therefore, software engineers must remember to follow best practices, stay disciplined, and be committed to producing quality test cases and software. These are just a handful of reasons why testing is vital to the Software Development Lifecycle (SDLC).